Unit-3 Syntax Analysis

* Finding First() and Follow()

First(A) contains all terminals present in first place of every string denoted by A

(1) S → abc | def | ghi
(2) First(terminal) = terminal
(3) First(ϵ) = ϵ

① S → ABc | ghi | jkl
A → a | b | c
B → b
D → d

First(S) = F(ABc), F(ghi), F(jkl)
= F(A), g, j = a, b, c, g, j
First(A) = a, b, c
First(B) = b
First(D) = d

② S → ABC
A → a | b | ϵ
B → c | d | ϵ
C → e | f | ϵ

ϵ ϵ
First(S) = F(ABC) = F(A), F(B, C), F(C)
= a, b, c, d, e, f, ϵ
First(A) = a, b, First(e) = a, b, ϵ
First(B) = c, d, e
First(C) = e, f, ϵ

③ $S \to ABC$

$A \to a \mid b$

$B \to c \mid d \mid \epsilon$

$C \to e \mid f \mid \epsilon$

First (S) = First (ABC) = First (A) = a, b

First (A) = a, b

First (B) = c, d, ε

First (C) = e, f, ε

④ $S \to ABC$

$A \to a \mid b \mid \epsilon$

$B \to c \mid d \mid \epsilon$

$C \to e \mid f$

First (S) = First (ABC) = a, b, First (Bc)

= a, b, e, d, First (c) = a, b, c, d, e, f

First (A) = a, b, ε

First (B) = c, d, ε

First (c) = e, f

⑤ $S \to ABC$

$A \to a \mid b \mid \epsilon$

$B \to c \mid d$

$C \to e \mid f \mid \epsilon$

First (S) = First (ABC) = a, b, First (Bc)

= a, b, c, d

First (A) = a, b, ε

First (B) = c, d

First (C) = e, f, ε

⑥

	First
$E \to TE'$	id, (
$E' \to *TE' \mid \epsilon$	*, ϵ
$T \to FT'$	id, (
$T' \to \epsilon \mid +FT'$	ϵ, +
$F \to id \mid (E$	id, (

First (E) = First (TE') = First (T) = id, (
= id, (

First (E') = *, ϵ = *, ϵ

First (T) = First (FT') = First (F)
= id, (= id, (

First (T') = ϵ, + = +, ϵ

First (F) = id, (= id, (

Terminal : Small letter (a-z), +, (,)

Non-terminal: A-Z (capital letters)

Finding First ( ) & Follow ( )

Follow (A) contains set of all terminals present in right of 'A'.

Rules 1) Follow of first symbol is $.

Fo(A) = { $ }

2) S → ACD

C → a | b

Follow (A) = First (C) = { a, b }

Follow (D) = Follow (S) = { $ }

3) S → aSbS | bSaS | ∈

Follow never contains ∈.

Follow(S) = b, a, $

S → AaAb | BbBa

A → ∈

B → ∈

Follow (A) = a, b

Follow (B) = b, a

S → ABC (∈ ∈) Follow(S) = { $ }

A → DEF Follow (A) = First (B) = First (C) = { $ }

B → ∈

C → ∈

D → ∈

E → F

F → ∈

		First	Follow
1.	$S \to aBDh$	$\{a\}$	$\{\$\}$
	$B \to cC$	$\{c\}$	$\{g, h, f\}$
	$C \to bC \mid \epsilon$	$\{b, \epsilon\}$	$\{g, f, h\}$
	$D \to EF$	$\{g, \epsilon, f\}$	$\{h\}$
	$E \to g \mid \epsilon$	$\{g, \epsilon\}$	$\{f, h\}$
	$F \to f \mid \epsilon$	$\{f, \epsilon\}$	$\{f, \epsilon\}$

$F(B) = F(D)$

		First	Follow
2.	$S \to ABC \mid CbB \mid Ba$	d, g, ϵ, h, a, b	$\{\$\}$
	$A \to da \mid BC$	d, g, ϵ, h	$\{h, g, \$\}$
	$B \to g \mid \epsilon$	g, ϵ	$\{h, \$, g\}$
	$C \to h \mid \epsilon$	$\{h, \epsilon\}$	$\{b, \$, g, h\}$

$S \to ACB \mid CbB \mid Ba$

$A \to aa \mid BC$

$B \to g \mid \epsilon$

$C \to h \mid \epsilon$

First (S) = First (ACB), First (CbB), First(Ba)
= d, g, h, First(CB), b, a = d, g, h, b, a

First (A) = d, First (B) = d, g, First(C)
= {d, g, h, ϵ}

First (B) = {g, ϵ}

First (C) = {h, ϵ}

		First	Follow
3.	$S \to ABC$	$\{a, b, c, d, e, f, \epsilon\}$	$\{\$\}$
	$A \to a \mid b \mid \epsilon$	$\{a, b, \epsilon\}$	$\{a, b, e, f, \$\}$
	$B \to c \mid d \mid \epsilon$	$\{c, d, \epsilon\}$	$\{e, f, \$\}$
	$C \to e \mid f \mid \epsilon$	$\{e, f, \epsilon\}$	$\{\$\}$

		First
4.	$S \to KMNOP$	
	$K \to R \mid \epsilon$	
	$M \to m \mid \epsilon$	$\{m, \epsilon\}$

		First	Follow
(4)	$E \to TE'$	{ (, id }	{ $,) }
	$E' \to +TE' \mid \epsilon$	{ +, ϵ }	
	$T \to FT'$	{ (, id }	
	$T' \to *FT' \mid \epsilon$	{ *, ϵ }	
	$F \to (E) \mid id$	{ (, id }	

(5)

be → be or bt | bt
bt → bt and bf | bf
bf → not bf | (b) | true | false

(5)

be → bt B'
B' → or bt B' | ϵ
bt → bf A'
A' → and bf A' | ϵ
bf → not bf | (be) | true | false

be → bt B'
B' → or bt B' | ϵ
bf → bf A'
A' → and bf A' | ϵ
bf → not bf | (be) | true | fa

E → E + T | T
T → T * F | F
F → id

→ E → TE'
E' → +TE' | ϵ
T → FT'
T' → *FT' | ϵ
F → (E) | id

Parsers

- Top down Parser
  - Reverse descent
  - *LL(1) or Predictive Parser (Imp)
- Bottom up parser
  - LR(k)
    - LR(0)
      - *LR(0) (Imp)
      - SLR(1) (Imp)
    - LR(1)
      - LALR (Imp)
      - CLR Imp
  - Operator Precedence

Parsing :- parsing is a process of deriving string from a given gramm

LL(1)
- Left to right Scanning
- Leftmost derivation
- 1 look ahead

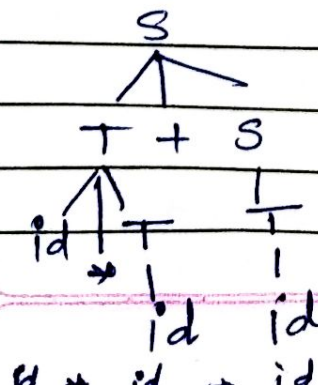
LR(0)
- Left to right Scanning
- Right most derivation in reverse orde

LR(0) < SLR < LALR < CLR (more powerful

$S \rightarrow T + S_{(1)} \mid T_{(2)}$

$T \rightarrow id * T_{(3)} \mid id_{(4)} \mid (S)_{(5)}$

String : id * id + id



LL(1) – Grammar Parsing Table

	First	Follow
$S \to (L) \mid a$ (1, 2)	{ (, a }	{ $, , ,) }
$L \to SL'$ (3)	{ (, a }	{) }
$L' \to \epsilon \mid ,SL'$ (4, 5)	{ , , ε }	{) }

	(	)	a	,	$
S	$S \to (L)$		$S \to a$		
L	$L \to SL'$		$L \to SL'$		
L'		$L' \to \epsilon$		$L' \to ,SL'$	

Predictive Parsing

Grammar

	First	Follow
$E \to TE'$	$F(E) = \{ id, (\}$	$F_o(E) = \{ \$,) \}$
$E' \to +TE' \mid \epsilon$	$F(E') = \{ +, \epsilon \}$	$F_o(E') = \{ \$,) \}$
$T \to FT'$	$F(T) = \{ id, (\}$	$F_o(T) = \{ \$,), + \}$
$T' \to *FT' \mid \epsilon$	$F(T') = \{ *, \epsilon \}$	$F_o(T') = \{ \$,), + \}$
$F \to (E) \mid id$	$F(F) = \{ id, (\}$	$F_o(F) = \{ \$, +, * \}$

Non Terminal	Terminal					
	id	+	*	(	)	$
E	TE'			TE'		
E'		$+TE'$			ϵ	
T	FT'			FT'		
T'		ϵ	$*FT'$		ϵ	ϵ
F	id			(E)		

All cell contain one and only one production

So grammar is LL(1).

STACK	INPUT	OUTPUT
$ E	id + id $	
$ E' T	id + id $	$E \rightarrow TE'$
$ E' T' F	id + id $	$T \rightarrow FT'$
$ E' T' id	id + id $	$F \rightarrow id$
$ E' T'	+ id $	
$ E'	+ id $	$T' \rightarrow \epsilon$
$ E' E' T +	+ id $	$T' \rightarrow +ET$
$ E' E' T	id $	
$ E' E' T' F	id $	$T \rightarrow FT'$
$ E' E' T' id	id $	$F \rightarrow id$
$ E' E' T'	$	
$ E' E'	$	$T' \rightarrow \epsilon$
$ E'	$	$E' \rightarrow \epsilon$
$	$	$E' \rightarrow \epsilon$

Handle & Handle Pruning

$S \rightarrow aABe$ i/p – abbcde

$A \rightarrow Abc \mid b$

$B \rightarrow d$

Handle during parsing of abbcde

Right Sential form	Handle	Reducing Prod^{n}
abbcde	b	$A \rightarrow b$
aAbcde	Abc	$A \rightarrow Abc$
aAde	d	$B \rightarrow d$
aABe	aABe	$S \rightarrow aABe$
S.		

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$ i/p – id * id

$F \rightarrow (E) \mid id$

Right sentinal forma	Handle	Reducing Production
id * id	id	$F \rightarrow id$
F * id	F	$T \rightarrow F$
T * id	id	$F \rightarrow id$
T * F	T * F	$T \rightarrow T * F$
T	T	$E \rightarrow T$
E	E	

Right most derivation in reverse order.

Operator Grammar

A grammar that is used to define mathematical operator is called an Operator grammar or operator precedence grammar.

A grammar is said to be operator prece-dence grammar if it has 2 properties:

i) No RHS of any production has a ϵ.
ii) No two non terminals are adjacent on RHS.

eg. $E = E+E \mid E*E \mid id$ // Operator grammar

eg) $S \rightarrow SAS \mid a$ // Not operator grammar
$A \rightarrow bSb \mid b$

We will convert it into operator grammar.

$S \rightarrow SbSbS \mid a \mid SbS \mid$
$A \rightarrow$ Not required

There are 3 operator precedence relations:

i) $a \doteq b \rightarrow$ means a has higher precedence than terminal b or a "takes precedence over" b.

ii) $a \lessdot b \rightarrow$ means a "yields precedence to" or a has lower precedence than terminal b.

iii) $a \doteq b \rightarrow$ means a has same precedence as b.

→ Bottom up parser that interprets an operator Grammar.

→ This parser is only used for Operator grammar.

→ Ambiguous grammar are not allowed in any parser except Operator precedence parser.

We will make Operator relation table or operator preeedence relation table for given grammar.

eg.) $E \rightarrow E+E \mid E*E \mid id$ + left asso.

	id	+	*	$
id	-	$\gtrdot$	$\gtrdot$	$\gtrdot$
+	$\lessdot$	$\gtrdot$	$\lessdot$	$\gtrdot$
*	$\lessdot$	$\gtrdot$	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	$\lessdot$	$\lessdot$

id > +

Two id's will never be compared because they will never come side by side.

Identifier will be given highest precedence compared to any other operator.

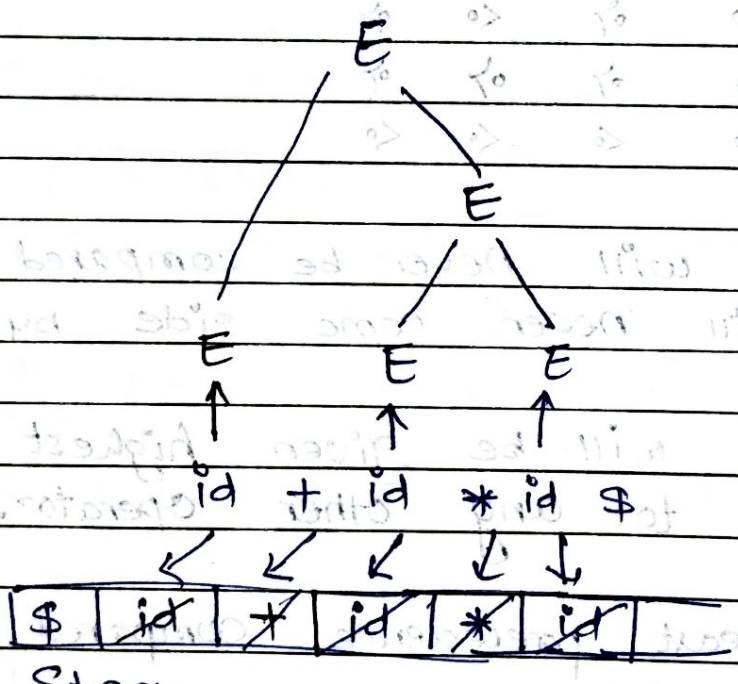
$ has least precedence compared to any other operator.

Using this operator precedence table parser will parse the i/p.

→ top of stack

i is $\lessdot$, push else pop i.e. reduce

Stack	input	Action
$	id + id * id $	push id
$ id	+ id * id $	pop - id. E→id
$ E	+ id * id $	push +
$ E +	id * id $	push
$ E + id	* id $	pop id E→id
$ E +	* id $	push
$ E + E	* id $	push
$ E + E *	id $	push id
$ E + E * id	$	
$ E + E * E	$	pop
$ E + E	$	
$ E		



Stack

$\$ \lessdot id$ Push

$id \gtrdot +$ pop

$\$ \lessdot +$ Push

$+ \lessdot id$ push

$id \gtrdot *$ pop

$+ \lessdot *$ push

$* \lessdot id$ push

$id \gtrdot \$$ pop

$* \gtrdot \$$ pop

$+ \gtrdot \$$ pop

Disadvantages of Operator Relation Table -
no. of enteries
i.e. if grammar has 4 operators than 16 entries.

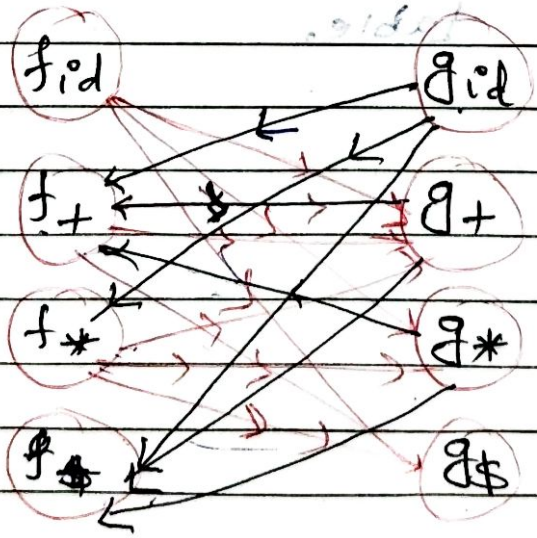
for N operator → $O(n^2)$

So, to decrease the size of table, we use operator fenction table.

f for rows and for column g.

	id	+	*	$
id	-	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	-

Now, construct a graph, for every operator, $f_{id}, g_{id}, f_+, g_+, f_*, g_*, f_\$, g_\$$.



fid → g+
fid → g*
fid → g$

gid ← f+
f+ → g+
f* ← g*
f+ → g$

if any cycle found in the graph, stop there because it is not possible to construct a operator in precendence fenction tabe

To make function table

To find out for every node that what is the length of the longest path start from that node.

$$fid \rightarrow g* \rightarrow f+ \rightarrow g+ \rightarrow f\$$$

$$gid \rightarrow f* \rightarrow g* \rightarrow f+ \rightarrow g+ \rightarrow f\$$$
$$\Rightarrow \quad 1 \quad 2 \quad 3 \quad 4 \quad 5$$

	id	+	*	$
f	4	2	4	0
g	5	1	3	0

Advantage → Size is less $O(2n)$

dis adv. → blank entry — errors

even though we have blank entries in relation table we get non blank entries in Fn Table.

So, error detecting capabilities of Fn Table < error detecting capability of relation table.

Check Operator grammar

P → SR | S
R → bSR | bS
S → Wbs | W
W → L * W | L
L → id

P → Sb(SR) | S
P → SbP | Sbs | S
P → SbS bS | S
S → Wbs | W
W → L * W | L
L → id

right recursive

Operator relation Table

	id	*	b	$
id	–	⋗	⋗	⋗
*	⋖	⋖	⋗	⋗
b	⋖	⋖	⋖	⋗
$	⋖	⋖	⋖	–

id at last level – highest precedence

* is defined as right recursive :-
it is right associative

LEX TOOL

Lex is a tool / computer programmar which generates Lexical analyzer. It is used with YACC programar / generator.

Lexical analyzer - 1st phase of compiler which generates several code in HLL. Streams of tokens.

Lex written by by mike Lesk and Eric Schmidst and described in 1975.

Lex Compiler

Lex Source → Lex → lex.yy.c
Program compiler
File.l

lex.yy.c → C compiler → a.out

i/p stream → a.out → stream of tokens
or HLL

1) Source code is in Lex language with filename.l extension.
It is given to Lex compiler which is the lex tool, and produce lex.yy.c — a C program as output.

2) C compiler runs this C code. i.e. lex.yy.c program and produce an output a.out

3) a.out transform an i/p stream into a sequence / stream of Tokens.

Structure of LEX or LEX File Format

A lex program is seperated into 3 Sections by %% delimeters.

```
% {declaration} // declaration of variable
%%                Including files / Libraries
% {Translation Rules} //- contains rules of
                          in form of Regular
                          expression
% {Auxillary functions}
```

eg.
```
%{  #include <stdio.h>      } declaration
      int c=0;
%}
%%
pattern {action}
%%

main ()
  {
  }
```

These expression we can write in Lex Pro

Rules :

abc - match abc

[a-z] - match small a to z.

[a-z]* - match all the strings in small letters with null or more than 1 character

[a-z]+ - 1 or more characters

[A-z a-z]+ -

^a means a should come at start

a$ means a at end

[0-9]+ 1 or more digits

YYlex() - reads the i/p stream and generates token a/c to the regular expression written in rules section.

YYwrap() - called by lex tool when i/p is exhausted
return 1 if i/p is finished
else 0.

YYtext - pointer to the i/p string.

Programs in LEX language

eg.

```
%{
    #include <stdio.h>
%}
%%
"hi"  { printf("By"); }      /* pattern — action (will be C lang. syntax) */
.*    { printf("wrong"); }
%%
main()
{
  printf("enter i/p");
  yylex();        — Take i/p & generates the token a/c to patterns in rule section
}
int yywrap()      — to identify end of i/p
{
  return 1;
}
```

ex. 2 Check odd or even

```
%{ #include <stdio.h>
        /* we can have comments */
    int m;
%}
%%
[0-9]+
    { m = atoi(yytext);
      if (m%2 == 0)
          pf("even");
      else
          Printf("odd");
      }
.*  { printf("wrong not a no."); }
%%
```

```
int yywrap( )
{
  return 1;
}

int main( )
{
printf(" enter the i/p");
yylex( );
return 0;
}
```

Bottom up Parser

↓

LR Parser

↓

LR(0) SLR(1) LALR(1) CLR(1)

i/p buffer → parser → parsing Table (Imp)

↓

stack

→ Construction of parsing Table will be differ

LR(0) item — LR(0), SLR

E → .9A

LR(1) item — LALR(1), CLR(1)

E → .9A, 9/b

LR Parsing

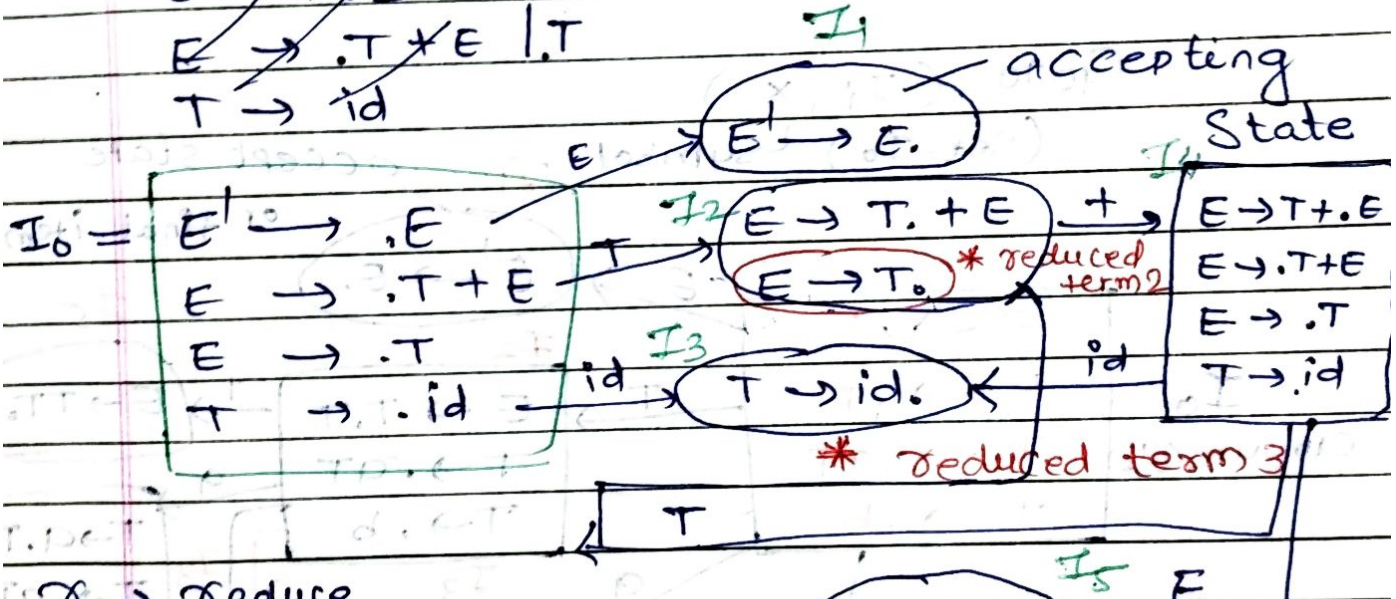
How to create LR Parsing table? OR weather given grammar accepted by LR or not.

(1) (2)
$E \to T+E \mid T$
$T \to id$ (3)

LR(0) CI
Canonical Item

Augment grammar

$E' \to .E$
$E \to .T+E \mid .T$
$T \to .id$



r → reduce
S → shift

r_2/S_4 → Shift-reduce parsing → SLR

State	Action			GOTO	
	id	+	$	E	T
0	S_3			1	2
1	-	-	Ac	-	-
2	r_2	S_4/r_2	r_2		
3	r_3	r_3	r_3		
4	S_3			5	2
5	r_1	r_1	r_1		

	id	+	$
0	S_3		
1			Accept
2		r_2	r_2
3		r_3	r_3
4			
5			r_1

LR(0)

$E \rightarrow TT$
$T \rightarrow aT / b$

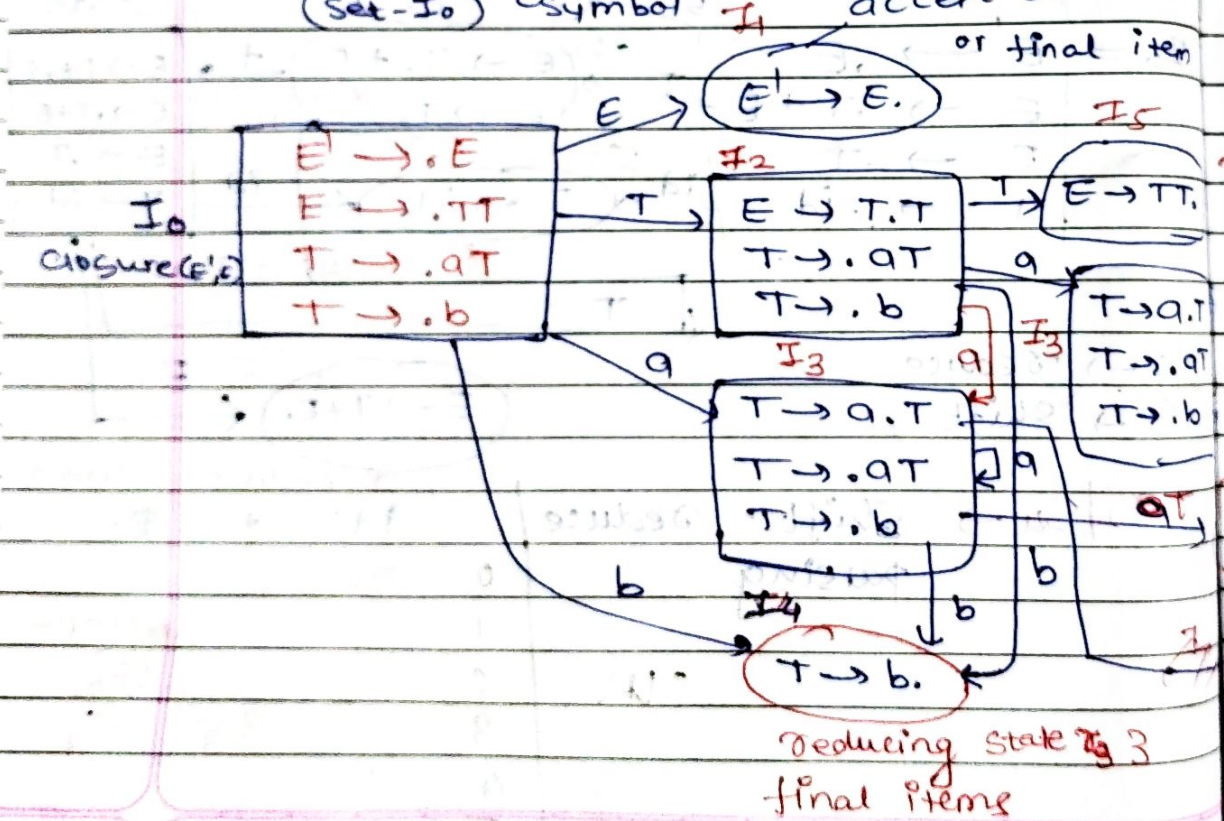
↓

Convert into augmented grammar by adding augmented production

$E' \rightarrow E$ (1)
$E \rightarrow TT$
$T \rightarrow aT / b$
(2) (3)

$E' \rightarrow .E$ (..right immediat if you found Nonterminal encude all)
$E \rightarrow .TT$
$T \rightarrow .aT$
. .

Goto (I_i, X)
(Set-I_0) Symbol



LR(0)

State
0
1
2
3
4
5
6

LR(0)

State	Action a	Action b	Action $	GOTO E	GOTO T
0	S_3	S_4		1	2
1			accept		
2	S_3	S_4			5
3	S_3	S_4			6
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

reducing state 1

I_6

$T \to aT.$

reducing state 2

$T \to aT.$

I_5

SLR parsing :

Grammar :

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

1) * Make augmented grammar

~~Augmented~~ grammar

$E' \rightarrow E$ // Add new start Symbol E'

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

2) Give numbers to production (will need it in table generation)

(0) $E' \rightarrow E$

(1) $E \rightarrow E+T$

(2) $E \rightarrow T$

(3) $T \rightarrow T*F$

(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

(6) $F \rightarrow id$

3) Find follow set for all non-terminal

Follow $(E) = \{ \$,), + \}$

Follow $(T) = \{ \$,), +, * \}$

Follow $(F) = \{ \$,), +, * \}$

4) Construct LR(0) item sets

$I_0 = E' \rightarrow .E$ // $E' \rightarrow .E$ has .E add production of E

$E \rightarrow .E+T$ $E \rightarrow .T$ " .T " " of T

$E \rightarrow .T$ $T \rightarrow .F$ " .F " " of F.

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

// After . there are 5 possibilities ETF(& prepare goto for all

$I_1 =$ goto $(I_0, E) = E' \rightarrow E.$ // Production with ".E" become ".E"

$E \rightarrow E.+T$

its new item set so give new name I_1 Same way when find new item set give new name

$I_2 =$ goto $(I_0, T) = E \rightarrow T.$

$= T \rightarrow T.*F$

$I_3 =$ goto $(I_0, F) = T \rightarrow F.$

$I_4 =$ goto $(I_0, ( ) = F \rightarrow (.E)$ // After . there is E (nonterminal) So add production of E

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow id$

I_5 = goto (I_0, id) = $F \to id.$

I_6 = goto (I_1, +) = $E \to E+.T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

I_7 = goto (I_2, *) = $T \to T*.F$
$F \to .(E)$
$F \to .id$

I_8 = goto (I_4, E) = $F \to (E.)$
$E \to E.+T$

I_2 = goto (I_4, T) = $E \to T.$
$T \to T.*F$

// Its same is I2 so don't give new name, give name I2 same when item match with any previous item set give same name

~~I_9 = goto (I_4, E) = $F \to (E.)$~~
~~$E \to E.+T$~~

~~I_2 = goto (I_4, T) = $E \to T.$~~
~~$T \to T.*F$~~

I_3 = goto (I_4, F) = $T \to F.$

I_4 = goto (I_4, () = $F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

I_5 = goto (I_4, id) = $F \to id$

I_9 = goto (I_6, T) = $E \to E+T$
$T \to T.*F$

I_3 = goto (I_6, F) = $T \to F.$

I_4 = goto (I_6, () = $F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

I_5 = goto (I_6, id) = $F \to id$

I_{10} = goto (I_7, F) = $T \to T*F$

I_4 = goto (I_7, () = $F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$

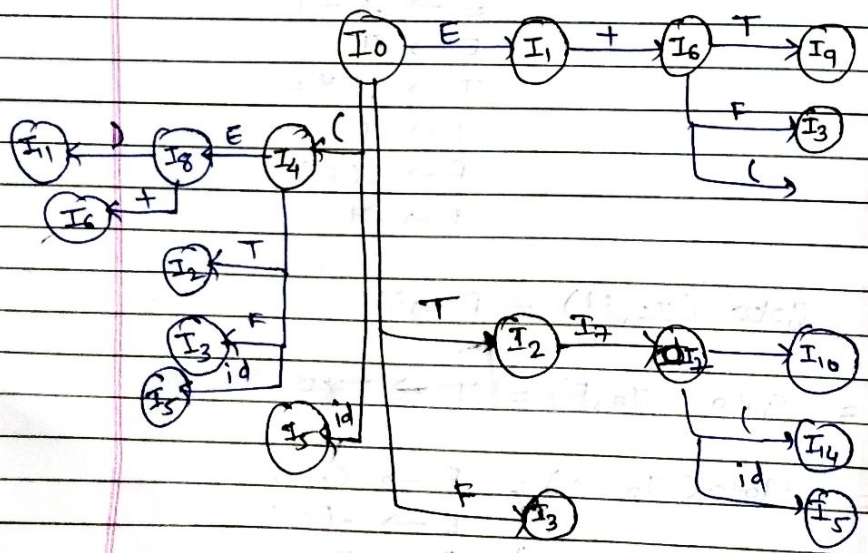
F → .(E)
F → .id

I_5 = goto (I_4, id) = F → id

I_{11} = goto (I_8,)) = F → (E).

I_6 = goto (I_8, +) = E → E+.T
T → .T*F
T → .F
F → .(E)
F → .id

I_7 = goto (I_9, *) = T → T*.F
F → .(E)
F → .id



Item set	action						goto		
	id	+	*	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6				Accept			
2		R_2	S_7		R_2	R_2			
3		R_4	R_4		R_4	R_4			
4	S_5			S_4			8	2	3
5		R_6	R_6		R_6	R_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		R_1	S_7		R_1	R_1			
10		R_3	R_3		R_3	R_3			
11		R_5	R_5		R_5	R_5			

	STACK	INPUT	ACTION
0	0	id * id + id $	shift
1	0 id 5	* id + id $	reduce F → id
2	0 F 3	* id + id $	reduce T → F
3	0 T 2	* id + id $	shift
4	0 T 2 * 7 id 5	+ id $	shift
5	0 T 2 * 7 F 10	+ id $	reduce F → id
6	0 T 2	+ id $	reduce T → T * F
7			
8			
9			
10			
11			

	Stack	Symbol	input	Action
	0		id * id + id $	Shift
(S5)	05	id	* id + id $	reduce by F→id
5*	03	F	* id + id $	reduce by T→F
3*	02	T	* id + id $	Shift
2*S7	027	T*	id + id $	Shift
7id-S5	0275	T*id	+ id $	reduce F→id
5+	0270	T*F	+ id $	reduce T→T*
10+	02	T	+ id $	reduce E→T
2+	01	E	+ id $	Shift
1+	0165	E+ id	$	Shift
6id	0163	E+F	$	reduce F→id
6F	0169	E+T	$	reduce T→F
3$	01	E	$	reduce E→E+T
0E				accept

	Stack	input	Action
	0	id * id + id $	Shift
	0 id 5 (Pop-5)	* id + id $	reduce F→id
check 0-F	0 F 3	* id + id $	reduce T→F
check 0-T	0 T 2	* id + id $	Shift
	0 T 2 *	id + id $	Shift
	0 T 2 * 7 id 5	+ id $	reduce F→id
check 7→T	0 T 2 * 7 F 10	+ id $	reduce T→T*F
	0 T 2	+ id $	reduce E→T
	0 E 1	+ id $	Shift
	0 E 1 + 6	id $	Shift
	0 E 1 + 6 id 5	$	reduce F→id
	0 E 1 + 6 F 5	$	reduce T→F
	0 E 1 * 6 T 9	$	E→E+T
	0 E 1	$	accept

Check 0 row id column S_5 – Shift id & 5

Check 5th row * column R_6 – reduce F→id, POP element

3 row – * column R_4 T→F

2nd row – * column Shift * S_7

7th row – Id || S_5

5th row – + Column reduce 6 F→id

10th row + reduce T→T*F r_3

CLR Parsing Table LR(1) Canonical item

$S \to aAd \mid bBd \mid aBe \mid bAe$

$A \to c$ (small c)

$B \to c$ (small c)

accepting state

0 $S' \to .S, \$$

1 $S \to .aAd, \$$

2 $S \to .bBd, \$$

3 $S \to .aBe, \$$

4 $S \to .bAe, \$$

S → $S' \to S.\$$

a → $S \to a.Ad, \$$; $A \to .c, d$; $S \to a.Be, \$$; $B \to .c, e$ — First

b → $S \to b.Bd, \$$; $S \to b.Ae, \$$; $B \to .c, d$; $A \to .c, e$

A → $S \to aA.d, \$$ → d → $S \to aAd., \$$ * reduce term 1

B → $S \to aB.e, \$$ → e → $S \to aBe., \$$ * reduce term 3

c → $A \to c.d$; $B \to c.e$ *

B → $S \to bB.d, \$$ → B → $S \to bBd., \$$ * reduce term 2

A → $S \to bA.e, \$$ → A → $S \to bAe., \$$ * reduce term 4

c → $B \to c., d$; $A \to c., e$ *

→ Write reduce term at lookhead symbol

LR(1) = LR(0) + 1 look ahead Symbol

$S \to AA$
$A \to aA \mid b$

$S' \to S$
$S \to AA$
$A \to aA \mid b$

LR(0) item	LR(1)
$S' \to .S$	$S' \to .S, \$$ First of $\$ = \$$
$S \to .AA$	$S \to .AA, \$$
$A \to .aA$	$A \to .aA, a/b$
$A \to b$	$A \to .b, a/b$

ex. $S \to .aA, bc, \$$
$A \to .b, b$ First of $bc, \$$

$S \to AA$
$A \to aA \mid b$

$S' \to S$
$SA \to AA$
$A \to aA \mid b$

I_0
$S' \to .S, \$$
$S \to .AA, \$$
$A \to .aA, a/b$
$A \to .b, a/b$

I_1
$S' \to S., \$$

I_2
$S \to A.A, \$$
$A \to .aA, \$$
$A \to .b, \$$

I_3
$A \to a.A, a/b$
$A \to .aA, a/b$
$A \to .b, a/b$

I_4
$A \to b., a/b$
* reduce term 3

I_5 * reduce term 1
$S \to AA., \$$

I_6
$A \to a.A, \$$
$A \to .aA, \$$
$A \to .b, \$$

I_7
$A \to b., \$$
* reduce term 3

I_8
$A \to aA., a/b$

I_9
$A \to aA., \$$

SLR

I_1
$S \to S., \$$

$A \to .AA, \$$
$S \to A.A, \$$
$A \to .aA, a/b$
$A \to .b, a/b$
lookahead

Action				Goto	
	a	b	$	S	A
0	S_3	S_4		1	2
1			Accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

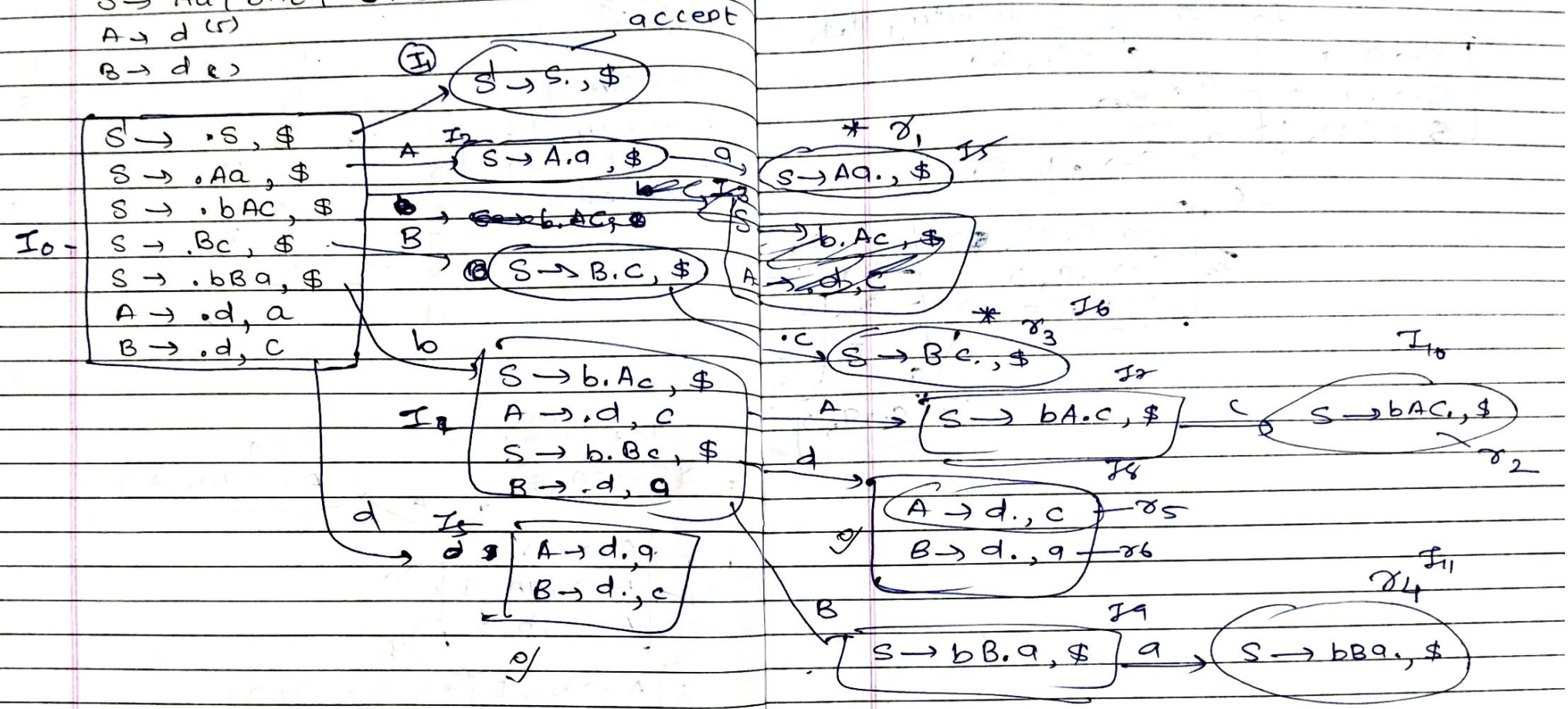
LALR

(1) (2) (3) (4)

S → Aa | bAc | Be | bBa

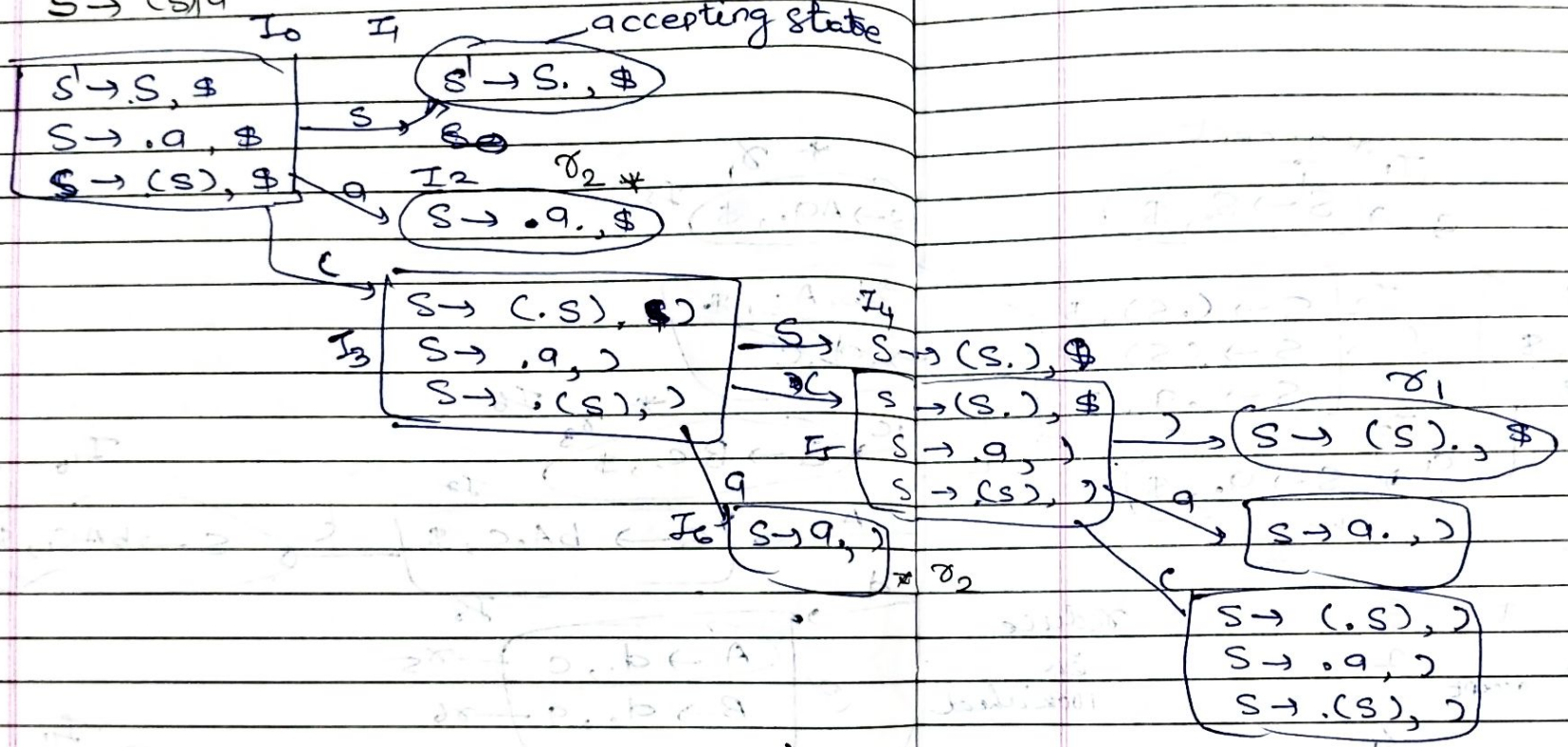
A → d (5)

B → d (6)



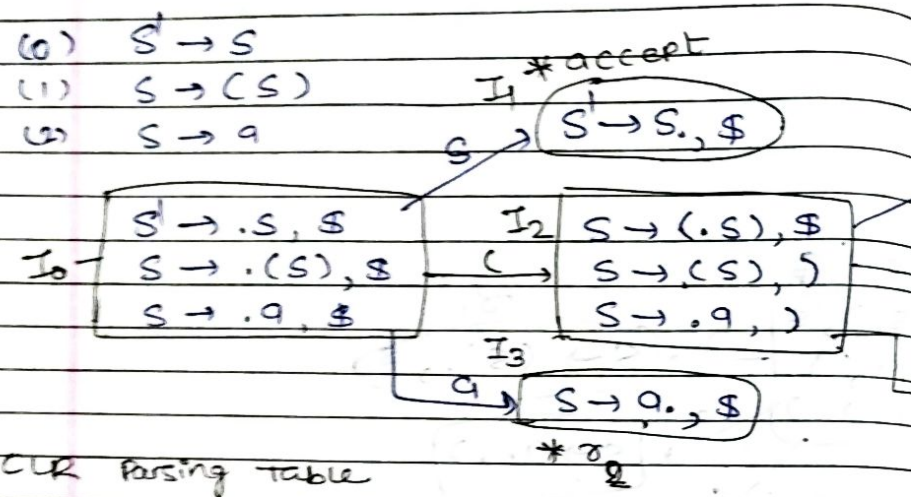
CLR VS LALR

$S \rightarrow (S) | a$



$S \rightarrow (S) \mid a$

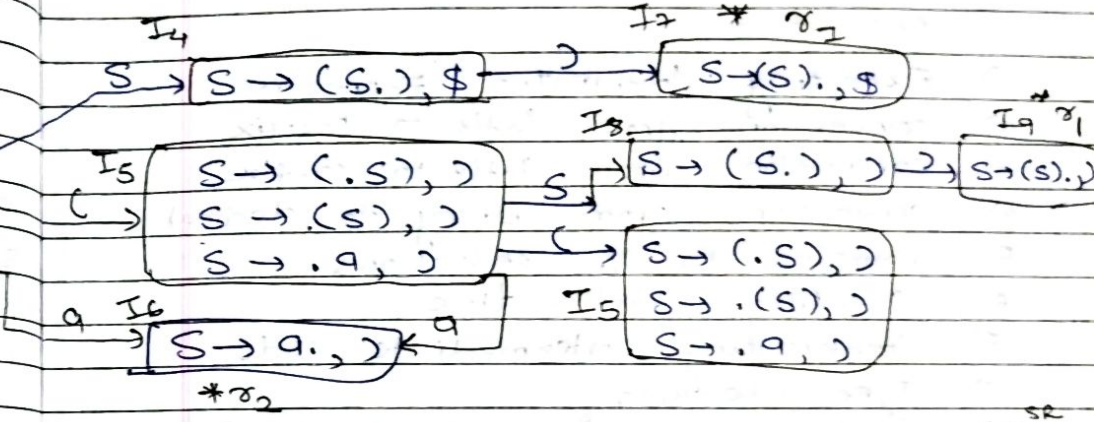
(0) $S' \rightarrow S$
(1) $S \rightarrow (S)$
(2) $S \rightarrow a$



CLR Parsing Table

	(	)	a	$	S	reduce on lookahead
0	S_2		S_3		1	
1				accept		
2	S_5		S_6		4	
3				r_2		
4						
5	S_5		S_6		8	
6		r_2				
7				r_1		
8		S_9				
9		r_1				

* CLR parsing — 10



LALR(1) parsing
— merge State

SR
LALR x
CLR SR x
CLR RR x
LLAR RR L

	(	)	a	$	S
0	S_2		S_3		1
1	S_5		S_6		4
25	S_5		S_6		48
36		r_2		r_2	
48		S_{79}			
79		r_1		r_1	